Lab Checkpoint 0: networking warmup

Due: Tuesday, April 11, 10 p.m.

Welcome to CS144: Introduction to Computer Networking. In this warmup, you will set up an installation of Linux on your computer, learn how to perform some tasks over the Internet by hand, write a small program in C++ that fetches a Web page over the Internet, and implement (in memory) one of the key abstractions of networking: a reliable stream of bytes between a writer and a reader. We expect this warmup to take you between 2 and 6 hours to complete (future labs will take more of your time). Three quick points about the lab assignment:

- It's a good idea to read the whole document before diving in!
- Over the course of this 8-part lab assignment, you'll be building up your own implementation of a significant portion of the Internet—a router, a network interface, and the TCP protocol (which transforms unreliable datagrams into a reliable byte stream). *Most weeks will build on work you have done previously,* i.e., you are building up your own implementation gradually over the course of the quarter, and you'll continue to use your work in future weeks. This makes it hard to "skip" a checkpoint.
- The lab documents aren't "specifications"—meaning they're not intended to be consumed in a one-way fashion. They're written closer to the level of detail that a software engineer will get from a boss or client. We expect that you'll benefit from attending the lab sessions and asking clarifying questions if you find something to be ambiguous and you think the answer matters. We'll update the "lab FAQ" document on the course website in response to late questions that need clarification.

0 Collaboration Policy

The programming assignments must be your own work: You must write all the code you hand in for the programming assignments, except for the code that we give you as part of the assignment. Please do not copy-and-paste code from Stack Overflow, GitHub, or other sources. If you base your own code on examples you find on the Web or elsewhere, cite the URL in a comment in your submitted source code.

Working with others: You may not show your code to anyone else, look at anyone else's code, or look at solutions from previous years. You may discuss the assignments with other students, but do not copy anybody's code. If you discuss an assignment with another student, please name them in a comment in your submitted source code. Please refer to the course administrative handout for more details, and ask on EdStem if anything is unclear. Services like GitHub Copilot or ChatGPT should be considered to be equivalent to "a student that took CS144 in a prior year."

EdStem: Please feel free to ask questions on EdStem, but please don't post any source code.

1 Set up GNU/Linux on your computer

CS144's assignments require the GNU/Linux operating system and a recent C++ compiler that supports the C++ 2020 standard. Please choose one of these three options:

1. **Recommended:** Install the CS144 VirtualBox virtual-machine image (instructions at https://stanford.edu/class/cs144/vm_howto/vm-howto-image.html).
2. Use a Google Cloud virtual machine using our class's coupon code (instructions at https://stanford.edu/class/cs144/vm_howto).
3. Run Ubuntu version 22.10, then install the required packages:

```
sudo apt update && sudo apt install git cmake gdb build-essential clang \
        clang-tidy clang-format gcc-doc pkg-config glibc-doc tcpdump tshark
```

4. Use another GNU/Linux distribution, but be aware that you may hit roadblocks along the way and will need to be comfortable debugging them. Your code will be tested on Ubuntu 22.10 LTS with `g++` 12.2 and must compile and run properly under those conditions.
5. If you have a 2020–23 MacBook (with the ARM64 M1 or M2 chips), VirtualBox will not successfully run. Instead, please install the UTM virtual machine software and our ARM64 virtual machine image from https://stanford.edu/class/cs144/vm_howto/.

2 Networking by hand

Let's get started with using the network. You are going to do two tasks by hand: retrieving a Web page (just like a Web browser) and sending an email message (like an email client). Both of these tasks rely on a networking abstraction called a *reliable bidirectional byte stream*: you'll type a sequence of bytes into the terminal, and the same sequence of bytes will eventually be delivered, in the same order, to a program running on another computer (a server). The server responds with its own sequence of bytes, delivered back to your terminal.

2.1 Fetch a Web page

1. In a Web browser, visit http://cs144.keithw.org/hello and observe the result.
2. Now, you'll do the same thing the browser does, by hand.
   (a) **On your VM**, run `telnet cs144.keithw.org http`. This tells the `telnet` program to open a reliable byte stream between your computer and another computer (named `cs144.keithw.org`), and with a particular *service* running on

that computer: the "`http`" service, for the Hyper-Text Transfer Protocol, used by the World Wide Web.[1]

If your computer has been set up properly and is on the Internet, you will see:

```
user@computer:~$ telnet cs144.keithw.org http
Trying 104.196.238.229...
Connected to cs144.keithw.org.
Escape character is '^]'.
```

If you need to quit, hold down ctrl and press], and then type `close` ↵.

(b) Type `GET /hello HTTP/1.1` ↵. This tells the server the *path* part of the URL. (The part starting with the third slash.)

(c) Type `Host: cs144.keithw.org` ↵. This tells the server the *host* part of the URL. (The part between http:// and the third slash.)

(d) Type `Connection: close` ↵. This tells the server that you are finished making requests, and it should close the connection as soon as it finishes replying.

(e) Hit the Enter key one more time: ↵. This sends an empty line and tells the server that you are done with your HTTP request.

(f) If all went well, you will see the same response that your browser saw, preceded by HTTP *headers* that tell the browser how to interpret the response.

3. **Assignment:** Now that you know how to fetch a Web page by hand, show us you can! Use the above technique to fetch the URL http://cs144.keithw.org/lab0/*sunetid*, replacing *sunetid* with your own primary SUNet ID. You will receive a secret code in the `X-Your-Code-Is:` header. Save your SUNet ID and the code for inclusion in your writeup.

2.2 Send yourself an email

Now that you know how to fetch a Web page, it's time to send an email message, again using a reliable byte stream to a service running on another computer.

1. SSH to *sunetid*@`cardinal.stanford.edu` (to make sure you are on Stanford's network), then run `telnet 148.163.153.234 smtp`.[2] The "`smtp`" service refers to the Simple Mail Transfer Protocol, used to send email messages. If all goes well, you will see:

```
user@computer:~$ telnet 148.163.153.234 smtp
```

[1] The computer's name has a numerical equivalent (`104.196.238.229`, an *Internet Protocol v4 address*), and so does the service's name (`80`, a *TCP port number*). We'll talk more about these later.

[2] These instructions might also work from outside Stanford's network, but we can't guarantee it.

```
Trying 148.163.153.234...
Connected to 148.163.153.234.
Escape character is '^]'.
220 mx0b-00000d03.pphosted.com ESMTP mfa-m0214089
```

2. First step: identify your computer to the email server. Type `HELO mycomputer.stanford.edu` ↵. Wait to see something like "`250 ... Hello cardinal3.stanford.edu [171.67.24.75], pleased to meet you`".

3. Next step: who is sending the email? Type `MAIL FROM: sunetid@stanford.edu` ↵. Replace *sunetid* with your SUNet ID.[3] If all goes well, you will see "`250 2.1.0 Sender ok`".

4. Next: who is the recipient? For starters, try sending an email message to yourself. Type `RCPT TO: sunetid@stanford.edu` ↵. Replace *sunetid* with your own SUNet ID. If all goes well, you will see "`250 2.1.5 Recipient ok`."

5. It's time to upload the email message itself. Type `DATA` ↵ to tell the server you're ready to start. If all goes well, you will see "`354 End data with <CR><LF>.<CR><LF>`".

6. Now you are typing an email message to yourself. First, start by typing the *headers* that you will see in your email client. Leave a blank line at the end of the headers.

```
354 End data with <CR><LF>.<CR><LF>
From: sunetid@stanford.edu ↵
To: sunetid@stanford.edu ↵
Subject: Hello from CS144 Lab 0! ↵
↵
```

7. Type the *body* of the email message—anything you like. When finished, end with a dot on a line by itself: `.` ↵. Expect to see something like: "`250 2.0.0 33h24dpdsr-1 Message accepted for delivery`".

8. Type `QUIT` ↵ to end the conversation with the email server. Check your inbox and spam folder to make sure you got the email.

9. **Assignment:** Now that you know how to send an email by hand to yourself, try sending one to a friend or lab partner and make sure they get it. Finally, show us you can send one to us. Use the above technique to send an email, from yourself, to `cs144grader@gmail.com`.

[3] Yes, it's possible to give a phony "from" address. Electronic mail is a bit like real mail from the postal service, in that the accuracy of the return address is (mostly) on the honor system. You can write anything you like as the return address on a postcard, and the same is largely true of email. Please do not abuse this—seriously. With engineering knowledge comes responsibility! Sending email with a phony "from" address is commonly done by spammers and criminals so they can pretend to be somebody else.

2.3 Listening and connecting

You've seen what you can do with `telnet`: a **client** program that makes outgoing connections to programs running on other computers. Now it's time to experiment with being a simple **server**: the kind of program that waits around for clients to connect to it.

1. In one terminal window, run `netcat -v -l -p 9090` **on your VM**. You should see:

   ```
   user@computer:~$ netcat -v -l -p 9090
   Listening on [0.0.0.0] (family 0, port 9090)
   ```

2. Leave `netcat` running. In another terminal window, run `telnet localhost 9090` (also on your VM).

3. If all goes well, the `netcat` will have printed something like "`Connection from localhost 53500 received!`".

4. Now try typing in either terminal window—the `netcat` (server) or the `telnet` (client). Notice that anything you type in one window appears in the other, and vice versa. You'll have to hit ⏎ for bytes to be transfered.

5. In the `netcat` window, quit the program by typing ctrl-C. Notice that the `telnet` program immediately quits as well.

3 Writing a network program using an OS stream socket

In the next part of this warmup lab, you will write a short program that fetches a Web page over the Internet. You will make use of a feature provided by the Linux kernel, and by most other operating systems: the ability to create a *reliable bidirectional byte stream* between two programs, one running on your computer, and the other on a different computer across the Internet (e.g., a Web server such as Apache or nginx, or the `netcat` program).

This feature is known as a *stream socket*. To your program and to the Web server, the socket looks like an ordinary file descriptor (similar to a file on disk, or to the `stdin` or `stdout` I/O streams). When two stream sockets are *connected*, any bytes written to one socket will eventually come out in the same order from the other socket on the other computer.

In reality, however, the Internet doesn't provide a service of reliable byte-streams. Instead, the only thing the Internet really does is to give its "best effort" to deliver short pieces of data, called *Internet datagrams*, to their destination. Each datagram contains some metadata (headers) that specifies things like the source and destination addresses—what computer it came from, and what computer it's headed towards—as well as some *payload* data (up to about 1,500 bytes) to be delivered to the destination computer.

Although the network tries to deliver every datagram, in practice datagrams can be (1) lost, (2) delivered out of order, (3) delivered with the contents altered, or even (4) duplicated and delivered more than once. It's normally the job of the operating systems on either end of the connection to turn "best-effort datagrams" (the abstraction the Internet provides) into "reliable byte streams" (the abstraction that applications usually want).

The two computers have to cooperate to make sure that each byte in the stream eventually gets delivered, in its proper place in line, to the stream socket on the other side. They also have to tell each other how much data they are prepared to accept from the other computer, and make sure not to send more than the other side is willing to accept. All this is done using an agreed-upon scheme that was set down in 1981, called the Transmission Control Protocol, or TCP.

In this lab, you will simply use the operating system's pre-existing support for the Transmission Control Protocol. You'll write a program called "`webget`" that creates a TCP stream socket, connects to a Web server, and fetches a page—much as you did earlier in this lab. In future labs, you'll implement the other side of this abstraction, by implementing the Transmission Control Protocol yourself to create a reliable byte-stream out of not-so-reliable datagrams.

3.1 Let's get started—fetching and building the starter code

1. The lab assignments will use a starter codebase called "Minnow." **On your VM**, run `git clone https://github.com/cs144/minnow` to fetch the source code for the lab.
2. Optional: Feel free to backup your repository to a **private** GitHub/GitLab/Bitbucket repository (e.g., using the instructions at https://stackoverflow.com/questions/10065526/github-how-to-make-a-fork-of-public-repository-private), but please make absolutely sure that your work remains private.
3. Enter the Lab 0 directory: `cd minnow`
4. Create a directory to compile the lab software: `cmake -S . -B build`
5. Compile the source code: `cmake --build build`
6. Outside the `build` directory, open and start editing the `writeups/check0.md` file. This is the template for your lab checkpoint writeup and will be included in your submission.

3.2 Modern C++: mostly safe but still fast and low-level

The lab assignments will be done in a contemporary C++ style that uses recent (2011) features to program as safely as possible. This might be different from how you have been asked to write C++ in the past. For references to this style, please see the C++ Core Guidelines (http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines).

The basic idea is to make sure that every object is designed to have the smallest possible public interface, has a lot of internal safety checks and is hard to use improperly, and knows how to clean up after itself. We want to avoid "paired" operations (e.g. malloc/free, or new/delete), where it might be possible for the second half of the pair not to happen (e.g., if a function returns early or throws an exception). Instead, operations happen in the constructor to an object, and the opposite operation happens in the destructor. This style is called "Resource acquisition is initialization," or RAII.

In particular, we would like you to:

- Use the language documentation at https://en.cppreference.com as a resource. (We'd recommend you avoid `cplusplus.com` which is more likely to be out-of-date.)
- Never use `malloc()` or `free()`.
- Never use **new** or **delete**.
- Essentially never use raw pointers (`*`), and use "smart" pointers (`unique_ptr` or `shared_ptr`) only when necessary. (You will not need to use these in CS144.)
- Avoid templates, threads, locks, and virtual functions. (You will not need to use these in CS144.)
- Avoid C-style strings (`char *str`) or string functions (`strlen()`, `strcpy()`). These are pretty error-prone. Use a `std::string` instead.
- Never use C-style casts (e.g., `(FILE *)x`). Use a C++ `static_cast` if you have to (you generally will not need this in CS144).
- Prefer passing function arguments by `const` reference (e.g.: `const Address & address`).
- Make every variable `const` unless it needs to be mutated.
- Make every method `const` unless it needs to mutate the object.
- Avoid global variables, and give every variable the smallest scope possible.
- Before handing in an assignment, run `cmake --build build --target tidy` for suggestions on how to improve the code related to C++ programming practices, and `cmake --build build --target format` to format the code consistently.

On using Git: The labs are distributed as Git (version control) repositories—a way of documenting changes, checkpointing versions to help with debugging, and tracking the provenance of source code. **Please make frequent small commits as you work, and use commit messages that identify what changed and why.** The Platonic ideal is that each commit should compile and should move steadily towards more and more tests passing. Making small "semantic" commits helps with debugging (it's much easier to debug if each commit compiles and the message describes one clear thing that the commit does) and protects you against claims of cheating by documenting your steady progress over time—and it's a useful skill that will help in any career that includes software development. The graders will be reading your commit messages to understand how you developed your solutions to the labs. If you haven't learned how to use Git, please do ask for help at the CS144 office hours

or consult a tutorial (e.g., https://guides.github.com/introduction/git-handbook). Finally, you are welcome to store your code in a **private** repository on GitHub, GitLab, Bitbucket, etc., but please **make sure your code is not publicly accessible.**

3.3 Reading the Minnow support code

To support this style of programming, Minnow's classes wrap operating-system functions (which can be called from C) in "modern" C++. We have provided you with C++ wrappers for concepts we hope you're familiar with from CS 110/111, especially sockets and file descriptors.

Please read over the public interfaces (the part that comes after "`public:`" in the files `util/socket.hh` and `util/file_descriptor.hh`. (Please note that a `Socket` is a type of `FileDescriptor`, and a `TCPSocket` is a type of `Socket`.)

3.4 Writing `webget`

It's time to implement `webget`, a program to fetch Web pages over the Internet using the operating system's TCP support and stream-socket abstraction—just like you did by hand earlier in this lab.

1. From the `build` directory, open the file `../apps/webget.cc` in a text editor or IDE.
2. In the **get_URL** function, find the comment starting "`// Your code here.`"
3. Implement the simple Web client as described in this file, using the format of an HTTP (Web) request that you used earlier. Use the `TCPSocket` and `Address` classes.
4. Hints:
   - Please note that in HTTP, each line must be ended with "\r\n" (it's not sufficient to use just "\n" or `endl`).
   - Don't forget to include the "Connection: close" line in your client's request. This tells the server that it shouldn't wait around for your client to send any more requests after this one. Instead, the server will send one reply and then will immediately end its outgoing bytestream (the one *from* the server's socket *to* your socket). You'll discover that your incoming byte stream has ended because your socket will reach "EOF" (end of file) when you have read the entire byte stream coming from the server. That's how your client will know that the server has finished its reply.
   - Make sure to read and print *all* the output from the server until the socket reaches "EOF" (end of file)—**a single call to `read` is not enough.**
   - We expect you'll need to write about ten lines of code.

5. Compile your program by running `make`. If you see an error message, you will need to fix it before continuing.

6. Test your program by running `./apps/webget cs144.keithw.org /hello`. How does this compare to what you see when visiting http://cs144.keithw.org/hello in a Web browser? How does it compare to the results from Section 2.1? Feel free to experiment—test it with any `http` URL you like!

7. When it seems to be working properly, run `cmake --build build --target check_webget` to run the automated test. Before implementing the **get_URL** function, you should expect to see the following:

```
$ cmake --build build --target check_webget
Test project /home/cs144/minnow/build
    Start 1: compile with bug-checkers
1/2 Test #1: compile with bug-checkers ........   Passed    1.02 sec
    Start 2: t_webget
2/2 Test #2: t_webget .........................***Failed    0.01 sec
Function called: get_URL(cs144.keithw.org, /nph-hasher/xyzzy)
Warning: get_URL() has not been implemented yet.
ERROR: webget returned output that did not match the test's expectations
```

   After completing the assignment, you will see:

```
$ cmake --build build --target check_webget
Test project /home/cs144/minnow/build
    Start 1: compile with bug-checkers
1/2 Test #1: compile with bug-checkers ........   Passed    1.09 sec
    Start 2: t_webget
2/2 Test #2: t_webget .........................   Passed    0.72 sec

100% tests passed, 0 tests failed out of 2
```

8. The graders will run your `webget` program with a different hostname and path than `make check_webget` runs—so make sure it doesn't *only* work with the hostname and path used by the unit tests.

4 An in-memory reliable byte stream

By now, you've seen how the abstraction of a *reliable byte stream* can be useful in communicating across the Internet, even though the Internet itself only provides the service of "best-effort" (unreliable) datagrams.

To finish off this week's lab, you will implement, in memory on a single computer, an object that provides this abstraction. (You may have done something similar in CS 110/111.) Bytes

are written on the "input" side and can be read, in the same sequence, from the "output" side. The byte stream is finite: the writer can end the input, and then no more bytes can be written. When the reader has read to the end of the stream, it will reach "EOF" (end of file) and no more bytes can be read.

Your byte stream will also be *flow-controlled* to limit its memory consumption at any given time. The object is initialized with a particular "capacity": the maximum number of bytes it's willing to store in its own memory at any given point. The byte stream will limit the writer in how much it can write at any given moment, to make sure that the stream doesn't exceed its storage capacity. As the reader reads bytes and drains them from the stream, the writer is allowed to write more. Your byte stream is for use in a *single* thread—you don't have to worry about concurrent writers/readers, locking, or race conditions.

To be clear: the byte stream is finite, but it can be *almost arbitrarily long*[4] before the writer ends the input and finishes the stream. Your implementation must be able to handle streams that are much longer than the capacity. The capacity limits the number of bytes that are held in memory (written but not yet read) at a given point, but does not limit the length of the stream. An object with a capacity of only one byte could still carry a stream that is terabytes and terabytes long, as long as the writer keeps writing one byte at a time and the reader reads each byte before the writer is allowed to write the next byte.

Here's what the interface looks like for the writer:

```
void push( std::string data ); // Push data to stream, but only as much as available capacity allows.

void close();     // Signal that the stream has reached its ending. Nothing more will be written.
void set_error(); // Signal that the stream suffered an error.

bool is_closed() const;              // Has the stream been closed?
uint64_t available_capacity() const; // How many bytes can be pushed to the stream right now?
uint64_t bytes_pushed() const;       // Total number of bytes cumulatively pushed to the stream
```

And here is the interface for the reader:

```
std::string_view peek() const; // Peek at the next bytes in the buffer
void pop( uint64_t len );      // Remove `len` bytes from the buffer

bool is_finished() const; // Is the stream finished (closed and fully popped)?
bool has_error() const;   // Has the stream had an error?

uint64_t bytes_buffered() const; // Number of bytes currently buffered (pushed and not popped)
uint64_t bytes_popped() const;   // Total number of bytes cumulatively popped from stream
```

Please open the `src/byte_stream.hh` and `src/byte_stream.cc` files, and implement an object that provides this interface. As you develop your byte stream implementation, you can run the automated tests with `cmake --build build --target check0`.

If all tests pass, the `check0` test will then run a speed benchmark of your implementation. Anything **faster than** `0.1 Gbit/s` (in other words, 100 million bits per second) is acceptable

[4] At least up to 2^{64} bytes, which in this class we will regard as essentially arbitrarily long

for purposes of this class. (It is possible for an implementation to perform faster than 10 Gbit/s, but this depends on the speed of your computer and is not required.)

For any late-breaking questions, please check out the lab FAQ on the course website or ask your classmates or the teaching staff in the lab session (or on EdStem).

What's next? Over the next four weeks, you'll implement a system to provide the same interface, no longer in memory, but instead over an unreliable network. This is the Transmission Control Protocol—and its implementations are arguably the **most prevalent computer program in the world**.

5 Submit

1. In your submission, please only make changes to `webget.cc` and the source code in the top level of `src` (`byte_stream.hh` and `byte_stream.cc`). Please don't modify any of the tests or the helpers in `util`.

2. Before handing in any assignment, please run these in order:

   (a) Make sure you have committed all of your changes to the Git repository. You can run `git status` to make sure there are no outstanding changes. Remember: make small commits as you code.

   (b) `cmake --build build --target format` (to normalize the coding style)

   (c) `cmake --build build --target check0` (to make sure the automated tests pass)

   (d) Optional: `cmake --build build --target tidy` (suggests improvements to follow good C++ programming practices)

3. Finish editing `writeups/check0.md`, filling in the number of hours this assignment took you and any other comments.

4. The mechanics of "how to turn it in" will be announced before the deadline.

5. Please let the course staff know ASAP of any problems at the lab session, or by posting a question on EdStem. Good luck and welcome to CS144!